Team Controller
Northern Arizona University
Flagstaff, Arizona
January 30th, 2024

Zachary Parham (Team Lead): zjp29@nau.edu
Tayyaba Shaheen (Mentor): ts2434@nau.edu
Bradley Essegian: bbe24@nau.edu
Brandon Udall: bcu8@nau.edu
Dylan Motz: djm658@nau.edu

Software Design

Northrop Grumman

Weapon System Support Software

Harlan Mitchell

Laurel Enstrom

Version: 1.5

Table of Contents

Table of Contents....2
1.0 Introduction....3
2.0 Implementation Overview....4
    2.1 QT Framework....4
    2.2 Git....5
    2.3 Main Branch....5
    2.4 Sprint Branch....5
    2.5 Issue Branches and Developer Branches....5
3.0 Architectural Overview....6
    3.1 Key Responsibilities & Features....6
        3.1.1 Controller....6
        3.1.2 Back End....7
        3.1.2 Log File....7
        3.1.3 GUI....7
        3.1.4 Installer....7
    3.2 Communication Mechanisms & Information Control....7
4.0 Module and Interface Descriptions....8
        4.0.1 Preface....8
    4.1 Connection....10
        4.1.1 Overview....10
    4.2 Electrical....11
        4.2.1 Overview....11
    4.3 Events....12
        4.3.1 Overview....12
    4.4 Controller Simulator (CSim)....13
        4.4.1 Overview....13
        4.5.2 CSim Structure and Management....14
        4.4.3 Serial Operations....15
    4.5 Data Display....15
        4.5.1 Overview....16
    4.6 Status....16
        4.6.1 Overview....17
5.0 Implementation Plan....17
6.0 Conclusion....18

1.0 Introduction

Our clients, Harlan Mitchell and Laurel Enstrom, represent Northrop Grumman. A cornerstone in the contracting space, Northrop Grumman has built everything from the B-2 Stealth Bomber to the James Webb Space Telescope. It's no surprise that these huge projects like these create hundreds of thousands of lines of diagnostic data, especially in the realm of weapon systems. In today's uneasy geopolitical climate, ensuring that our military armament systems are functioning optimally is not just a matter of national security; it is a global concern.

The weapon systems industry has witnessed a meteoric rise of the use of technology in recent years. Just like any industry in the 20th to 21st centuries, weapon systems saw the implementation of their processes controlled by software. In the early to mid 20th century, weapons were controlled and fired by a mechanism that was triggered by the gunner. Now, multi-billion dollar fighting machinery comes equipped with a weapon controller; a piece of technology that facilitates the firing and reloading of these weapons by the aforementioned software. Team Controller's job is to create a desktop application that allows engineers to easily identify issues within the weapon systems.

The existing solution to gather, parse, and extract valuable information from these weapon systems involves a lengthy process. The steps to this process include a chain of problem reporting from field operators to the engineers. If a problem is identified, an engineer will spend their valuable time traveling to the site where the fighting machine is located, with countless more on site visits in the future if the problem isn't solved.

To revolutionize the process of diagnosing and resolving issues with military defense systems, Team Controller and Northrop Grumman present a new solution—a simple, easy to use, and secure desktop application that implements a graphical user interface (GUI) to display and download diagnostic information. This product will be able to read diagnostic data via serial communication, directly interfacing with the weapon controller. This is similar to how an OBD2 scanner functions in the automotive industry. The main objective of our solution is to provide the end-users with the ability to easily collect diagnostic data without the need for dispatching engineers to their location.

Breaking down this goal, the team has created key user-level requirements. These include:

- The data display module shall be a desktop application;
- The data display module shall read input data via RS422 serial protocol from the controller simulator through a user specified port;
- The data display module shall have the ability to write event data into a log file.

The requirements above details the desktop application that displays the data from the controller on a GUI. For security reasons, the application will take in information through a serial communication port and the data retrieved will be accessible through a log file for ease of data manipulation and parsing.

2.0 Implementation Overview

The proposed solution is designed to solve the problem of engineers wasting time and resources by traveling to different testing sites. To accomplish this goal, the team has identified three core features the application must have. These are a serial communication connection between the application and the weapon controller, a graphical user interface or GUI, and the ability to produce a log file. The overall approach Team Controller will take includes the following software frameworks and technologies.

2.1 QT Framework

The main base on which our solution rests is the QT Framework. QT is the best choice for this application as it has built in serial communication and GUI development libraries. The installation and configuration of these packages is easily done through the QT Maintenance Tool that comes with the QT installation. The specific packages we installed through this tool are the QT Serial Bus and the QT Serial Port.

As well as serial communication, the QT Framework also assists in easy log file generation and saving. Using a built-in library called QT Dialog, a folder location can be chosen to save the log file to. Using this library with C++ ability to write in .txt or .csv format, makes this the best route to address the log files.

2.2 Git

Git and subsequently Github are industry standards for a reason. These tools allow the team to efficiently coordinate and solve problems. The generalized branch structure we are using goes as follows:

2.3 Main Branch

During the team's sprints, the main branch will not be touched. This branch holds working code as well as some documentation for our clients.

2.4 Sprint Branch

The sprint branch is named *sprint_x* with x being the sprint's number we are on. At the beginning of the sprint, the sprint branch is created off of the main branch. This branch will see all the sprints work being merged to it.

2.5 Issue Branches and Developer Branches

These branches are the working branches that the developer works on when solving issues. The key difference between the two are issue branches are generally used when two developers are working together and a single developer branch is just for that developer.

Once the issue or problem has been resolved, the developer will push their code, create a pull request and these pull requests will be approved by the rest of the team. Usually these pull requests are approved in a team call where the approvers can ask clarifying questions directly to the developer.

3.0 Architectural Overview

For the program's architecture, we have designed a simple diagram (Figure 1) that breaks down the system into several components. Similar to a model-view-controller architecture pattern, each component is a separate piece in which the back-end acts as a liaison between them all.

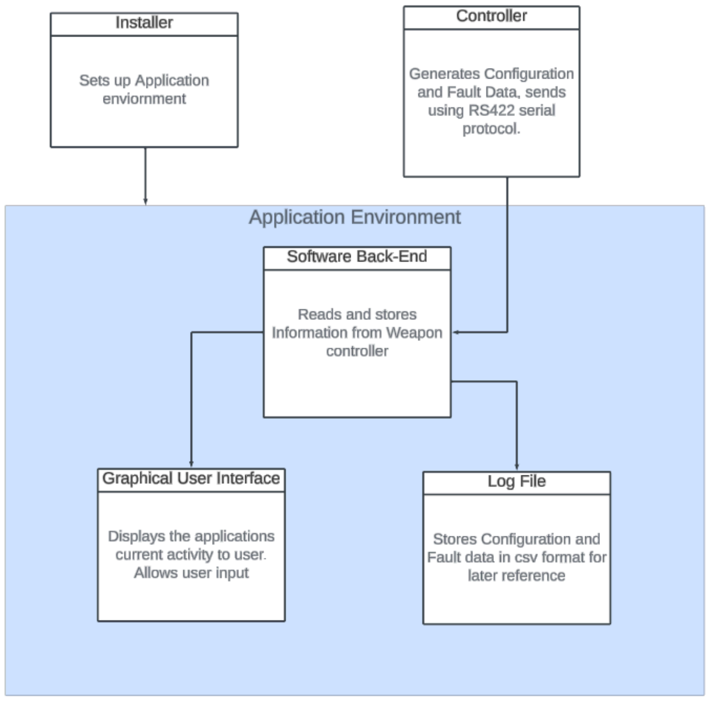

Installer
Sets up Application enviornment
Controller
Generates Configuration and Fault Data, sends using RS422 serial protocol.
Application Environment
Software Back-End
Reads and stores Information from Weapon controller
Graphical User Interface
Displays the applications current activity to user. Allows user input
Log File
Stores Configuration and Fault data in csv format for later reference

Figure 1 - Architecture Diagram

3.1 Key Responsibilities & Features

3.1.1 Controller

The weapon controller is responsible for supplying our program with the data it needs to update its GUI and log files. The controller's purpose is to manage the weapon and record any relevant weapon data recorded during operation. This data is then sent via RS422 serial communication to our program's back end.

3.1.2 Back End

The back-end component's main responsibility is to be able to create a connection with an attached serial device and read messages sent through that port. Another feature of the back-end is to figure out how to properly interpret those messages—which are separated into errors, events, status updates, and electrical components. Data is stored in associated classes and frequently updated as serial communication occurs.

3.1.2 Log File

The main responsibility of the log file component is to be able to download the configuration and fault data locally on the user's computer. The logfile should be generated each time the user requests on the GUI, as well as each time a controller session has completed (with up to five autosaves).

3.1.3 GUI

The GUI is the key component of our program, as it displays the data that has been processed to our users in one neat interface. Using the Qt framework, we will organize this data into an events/errors page, a status page, and an electrical page. There will also be a settings page that accepts input so that the user can customize their serial settings from the defaulted configuration.

3.1.4 Installer

The final component, the installer, will need to package and build our software as well as set up a file system to store log files and GUI resources. An Installer allows the program to be easily distributed and set up by Northrop Grumman engineers or customers.

3.2 Communication Mechanisms & Information Control

The way that each component of our system will interact with each other is quite simple, but follows a specific order for communication. The simulated **Controller** will *communicate with* the **Backend** by sending abstracted data via serial communication. We can use various library calls from the Qt framework to help set up the connection and to send/receive serial messages. From there, the data will be stored and manipulated to be more readable. The **Backend** can then *update* the **GUI** and **Log File** through the use of carefully constructed function calls with the information that has been received. The Qt library has an efficient approach for updating a UI referred to as “signals and slots”. Whenever a certain signal is emitted, such as when there is new data to be read, a slot is run to update the information in the GUI. As a requirement, this entire environment must be *set up by* the **Installer** component, accomplished by using the Inno tool.

4.0 Module and Interface Descriptions

Here we will break down the structure of each of the modules listed in section 3.0. Figure 2 (below) demonstrates how each class works together to support the overall system.

4.0.1 Preface

Signals and Slots: Our application uses a signal and slot infrastructure provided by Qt. To put it simply, signals are emitted as a result of certain events occurring during runtime. For example, if the user clicks the download button, a signal called download() is emitted. We can tie signals to slot functions, in this case the slot function is on_download_button_clicked(). So when the user emits the download() signal, our program calls the corresponding slot to process the request. In practice, this allows the program to be responsive in processing events without the need for an execution loop.

Serial Communication: Our clients have requested we use serial communication, specifically a serial protocol known as **RS422**. This is a type of wired connection between 2 devices using serial port hardware and a wire to connect the ports and transmit data between them. For testing we have virtualized this hardware using a product called Com0Com which allows us to operate using this technology virtually all on one device.

Terminology: Our application renders a **GUI** (graphical user interface) which displays data and allows for user interaction. The class responsible for managing the GUI is known as DataDisplay, or **DDM** for short. To simulate Northrop Grumman’s weapon controllers we have made a class called **CSim** which is short for controller simulator.

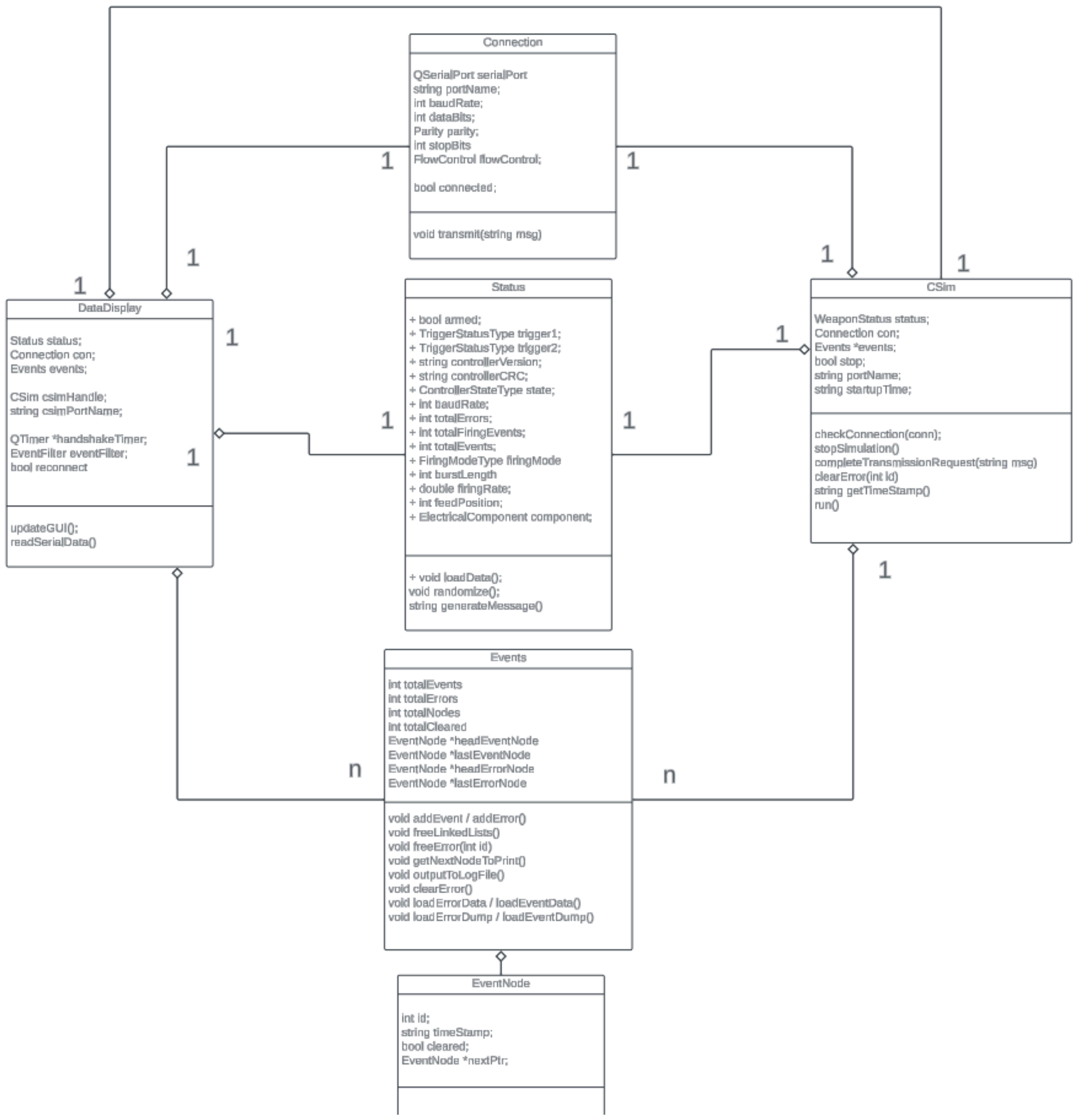


Figure 2 - Class Diagram

4.1 Connection

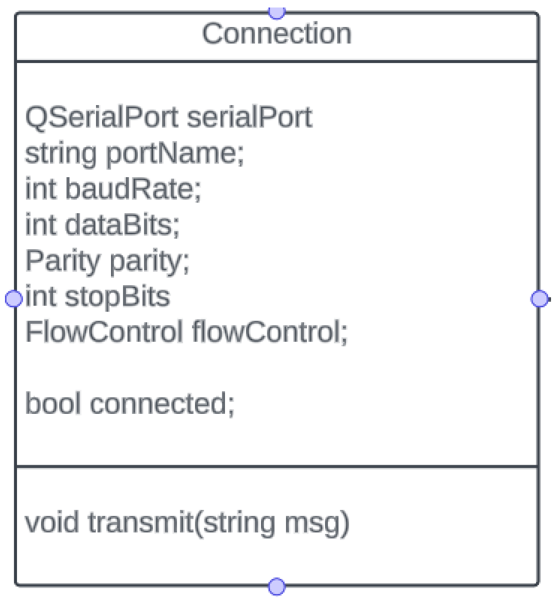


Figure 3 - Connection Class

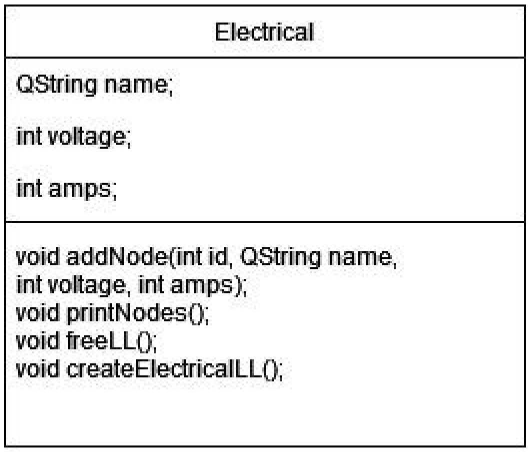
The purpose of the Connection class is to abstract serial communication operations under simple functions such as transmit(). This class is rather simple and it relies on QT's built in QSerialPort library which provides the infrastructure to configure serial settings and send/read messages from serial ports.

When declaring this class, the serial settings are provided to the constructor which then opens the serial port and configures it using the given settings. After that, any time a new message is detected in the serial port buffer, a signal named readyRead() is emitted which notifies the parent class that there is new data to be processed. This signal can be tied to a slot function within the parent class which can handle the processing. Both CSim and the DDM have separate functions connected to this signal (CSim has checkConnection() and DDM has readSerialData()).

The class contains a boolean which holds the connection status. This bool is to be updated within the parent class as a result of receiving a disconnect message or a handshake message.

Finally the transmit() function writes the given message to the serial port and waits for the data to be transmitted before returning. Error handling is built into this function as well to simplify the process for the parent class.

4.2 Electrical



```
Electrical
QString name;
int voltage;
int amps;
void addNode(int id, QString name,
int voltage, int amps);
void printNodes();
void freeLL();
void createElectricalLL();
```

Figure 4 - Electrical Class

Much like the CSim class mentioned in section 4.5, the electrical class' purpose is to simulate the data that might come in from electrical components located inside a weapon.

The class holds three variables: the name of the electrical component, the number of volts that component is using or producing, and the number of amps the component is using or producing. These values are initialized as empty when declaring the class. Because the team is using linked lists as the primary data structures, there is an additional electricalNode struct with the same variables.

The addNode, printNode, createElectricalLL and freeLL functions are all designed to work with the electricalNode linked list. To generate the electrical linked list, the function createElectricalLL is called which in turn calls addNode with randomized values for voltage and amps. The name of the electrical component is chosen at random from a list of electrical components. This will create a linked list of three or four components that will then get taken by the DDM to display to the user. The freeLL function is called by the deconstructor, and deletes the electricalNodes in the linked list.

4.3 Events

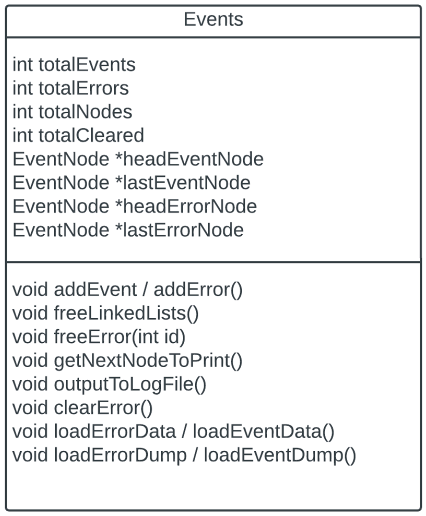


Figure 5 - Events Class

The Events component is a crucial part of the larger context of the program's architecture as it is responsible for managing and handling all serial messages that are determined to be "events". To keep things simple, the component manages two linked lists (of type EventNode, defined within the component) to store information about these messages; including IDs, timestamps, event strings, and its active/inactive status if it is an error. Since these are linked lists, each EventNode stores a pointer to the next node in the list.

The class also defines several variables that act as counters to store the total number of events, errors, cleared errors, and nodes to use when updating the GUI. To easily access the linked lists, a head pointer and last pointer are defined for both the Event list and Error list. The addEvent(), addError(), freeError(), clearError(), and freeLinkedLists() methods are all standard helper functions to modify the information in the linked lists.

Since logging these messages is a significant client requirement, the Events component will have functionality to output the linked lists to a readable log file. The method outputToLogFile() is self-defining, as it just loops through both linked lists and writes the data to a file based on ID. For modularity, getNextNodeToPrint() was defined as a separate helper function which simply looks into both linked lists and retrieves the next node in either list in order of ID. This method will also be useful when updating the GUI.

Finally, the loadErrorData(), loadEventData(), loadEventDump(), and loadErrorDump() methods are all used to update the linked lists with new nodes whenever the data display module reads new serial messages from the controller. They all use the addEvent() or addError() methods to add messages, in order, to the end of their respective linked list.

4.4 Controller Simulator (CSim)

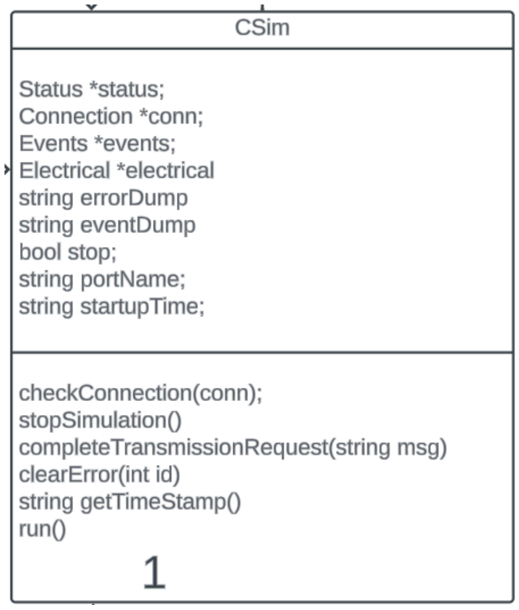


Figure 6 - CSim Class

4.4.1 Overview

Our application will be interfacing with Northrop Grumman's weapon controllers. We must be able to communicate with these controllers using RS422 serial communication. Due to the secretive nature of the weapon controller software, our team is not allowed access to the real weapon controllers. So we must develop a module capable of generating simulated weapon data and feeding it to our application through virtualized serial ports. That is where CSim comes in.

The CSim class must declare its own copy of several classes used in our DDM. These classes include Status, Connection, Electrical and Events. The idea is that CSim will randomly update these classes locally, then send update messages to the DataDisplay class, which is in charge of updating the GUI with this new data.

4.5.2 CSim Structure and Management

CSim is run within a thread, and it is managed mostly independently of the DDM. However, for testing purposes, we have included methods of managing certain aspects of CSim from our DDM. These include the completeTransmissionRequest() and clearError() slot functions. Using the GUI rendered by our DDM we can tell CSim to send custom messages through its serial port to be read by our DDM. This is useful for testing specific cases which might occur. The clearError() function is used to tell CSim to clear a specific error and send a clear error update through the serial port. Again this is useful for controlling the state of CSim for testing unique cases.

CSim's operations are conducted within a loop contained in the run() function. This function iterates at specified time intervals (the current interval is 2 seconds). Within each iteration, the status class is randomized with new values, then the program has a 20% chance of generating a new error and a 20% chance of generating a new event. There is also a 10% chance of clearing an error that was previously created with a not cleared flag. Upon completion, each of these operations must be reported to the DDM using a serial message, so after each operation a message is generated and sent through the serial port.

Each event and error message must include a timestamp which represents the point during the session when the event/error occurred. The getTimeStamp() function is responsible for comparing the current time with the startupTime and generating the time since startup timestamp.

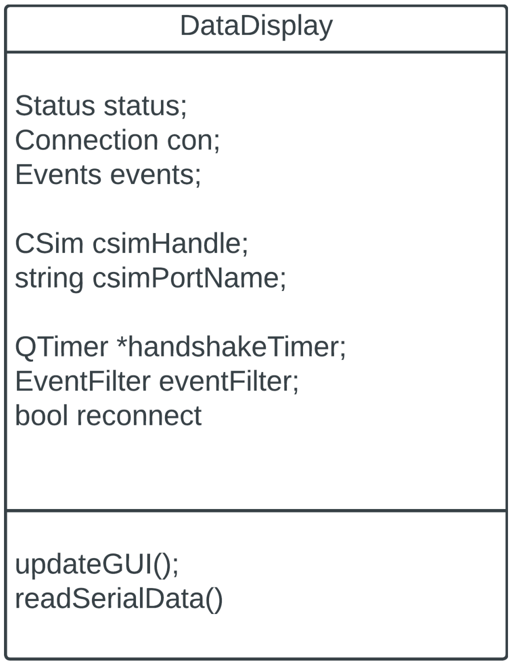
To stop CSim, the stopSimulation() function is used, which causes the run() function to exit its execution loop and clear all the data generated during the session. A disconnect message is then sent to the DDM and the thread is terminated.

4.4.3 Serial Operations

In the case that there is no active connection between CSim and our application, relevant messages (i.e. error and event messages) will be appended to a string known as a dump string. When a connection is made to the DDM, the error and event dumps are sent which allows the DDM to load all of the data that was created before the connection occurred and catch up to the controllers current state.

The checkConnection() function is used to evaluate the connection status between the DDM and CSim. It reads serial messages from the DDM, most importantly disconnect messages (used to end a connection) and handshake messages (used to start a connection).

4.5 Data Display



```
DataDisplay

Status status;
Connection con;
Events events;

CSim csimHandle;
string csimPortName;

QTimer *handshakeTimer;
EventFilter eventFilter;
bool reconnect

updateGUI();
readSerialData()
```

Figure 7 - DataDisplay Class

The data display class is very important as it is responsible for all the front-end work for the project. The goal of this class is to populate the front-end with all the data from the back-end which includes Connection, Electrical, Events, Status, and CSim.

The class also has many variables starting off with the first 4 which are the classes attached to Status, Connection, Events, and CSim. These variables will be used to gather the needed data to update the GUI with. Next we have the csimPortName which is testing purposes only for developers as it is the port name for the simulator. Next the timer variable keeps track of the time since the handshake started between the simulator and the program. Then the EventsFilter is an enumeration to determine which case for events/errors to be filtered and displayed on the GUI. Finally the reconnect boolean is a flag used to determine if the program is connected to the CSim.

Finally for this class we only have two functions which are updateGUI() and readSerialData(). The function updateGUI() updates the front-end whenever a new event, error, or status update comes in. Then readSerialData() gets the data from the back-end and stores the data in the correct variables used to update the front-end.

4.6 Status

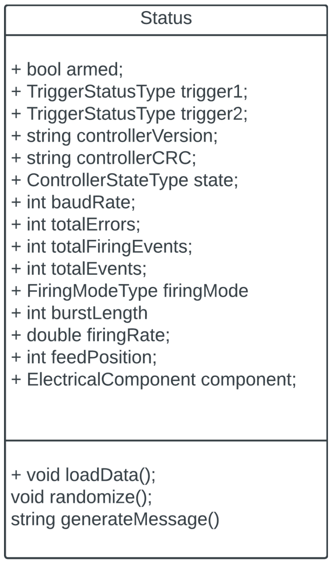


Figure 8 - Status Class

The status class is responsible for managing the current status of the simulated weapon.

The status class has the most variables since it needs a variable for each status of the weapon. For examples in the status class it needs to show if the weapon is currently armed(boolean), number of events and errors(int), the firing mode the weapon is in(enumeration), the current version and CRC of the controller(string), and the degree of the fieldPosition(int).

Finally the status class has three functions and this includes loadData(), randomize(), generateMessage(). The loadData() function breaks down the string containing all the needed status data and loads the data into each of the status classes variables. Then the randomize() function randomizes the values for the status. Finally the generateMessage() function concatenates all the status data into a string.

5.0 Implementation Plan

The team plans on using agile project management during implementation of the project. To start off our plan we are going to do 2 week long sprints. Each sprint has multiple meetings which start off on the first Monday and this meeting is to determine what task each team member is responsible for doing during that sprint. The tasks are tracked using github's issues tracker and when a team member finishes an issue they mark it as done and push their changes from their branch to the sprint branch. Then every Friday is the team's issue refinement meeting as this is to touch up any current issues and add any new issues needed for the project. Finally the second Monday is used to update the team of how the current sprint is going. Then at the end of each sprint the team will reflect on how the sprint went and push the sprint branch into the main branch on github.

The modules we have already completed by the time of writing this report are:

- CSim.cpp;
- Connection.cpp;
- Status.cpp;
- Events.cpp;
- Data Display.cpp.

These modules have most of the development infrastructure in place and only small additions and tweaks are needed. The rest of the modules, namely Electrical.cpp, have yet to be completed and are planned as top priorities in the coming sprints.

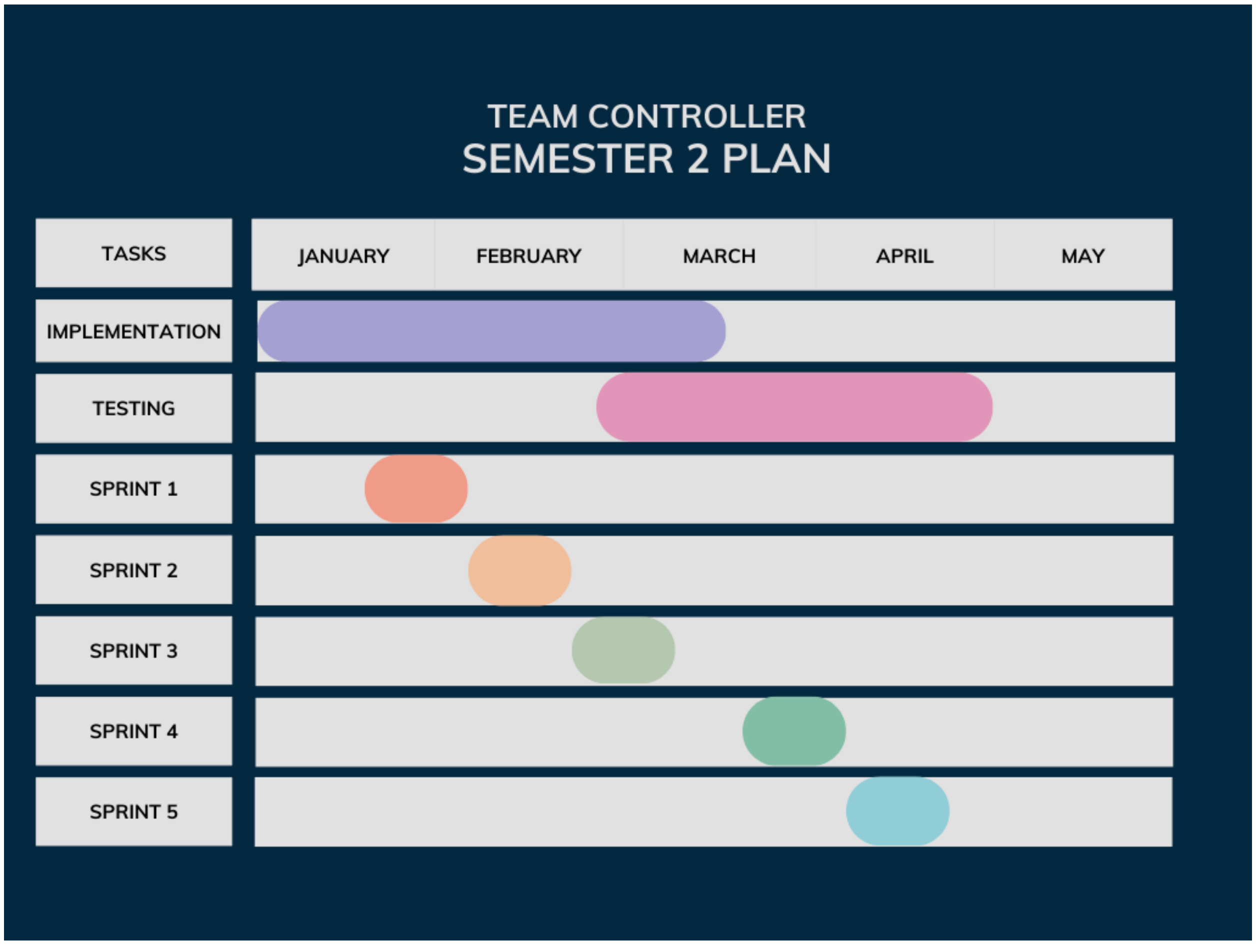


Figure 9 - Gantt Chart

As seen in the gantt chart the team started development over break which included the development of the CSim and GUI layout. Next during Sprint 1 the team plans on populating the events, and settings page of the GUI. For Sprint 2 the team will populate the electrical and status page of the GUI and research any issues (since the team is using new software we may need to understand how some libraries work). Then Sprint 3 will be used for testing, additional features, implementing researched issues. Finally Sprint 4 and 5 will be used for further testing, implementing unit tests, and polishing up the project.

6.0 Conclusion

Northrop Grumman's diagnostic capabilities will greatly improve with the use of this application. Not only would the engineer's time be saved but a large amount of company resources could be as well. With this application in use, engineers can easily view the inner workings of the weapon, see all events and errors generated by the controller and track down many bugs involved with weapon testing.

The desktop application will have four main components that will take the classes mentioned in section 4.0 to materialize:

- The GUI to show the engineer a visual representation of the weapon’s status;
- A log file download to allow for data analysis;
- An installer that would install the application without windows administrator rights;
- And the controller which is simulated for security reasons.

All of the above components will be handed over to the software engineers at Northrop Grumman to create a completed diagnostic application, with full weapon controller connectivity. This document will act as a guiding light for the team as we embark on the next step; development.